Analgesia Database: Caching library

Version 0.90

J.M. van Schalkwyk

February 27, 2009

Contents

1 The C file: cache.c

The menu structure of the pain database is somewhat hierarchical: we work by ward and then select an individual patient. This suggests that if we had some way of tweaking our tables so that only data for a particular patient were accessible, we might speed SQL data access considerably. We find that this is the case — performance is far better if we temporarily constrain the PROCESS table to refer to a single patient (say that patient with database ID number 1234), and even better if we subsequently constrain the EPOCH table to refer only to such processes. We do so by saying respectively:

```
CACHE(PROCESS.Person.1234)
```

. . . and . . .

```
CACHE(PROCESS:EPOCH.Process)
```

We can subsequently UNCACHE the tables in reverse order. This library is very similar to that described in *IdxLib.tex*.

1.1 Rationale

Examining things in more detail, the rationale for the existence of this library is threefold:

1. Up to 90% of PDA processor time is consumed by our SQL SELECT statements;
2. There are major problems with speed of SQL queries on the PDA, exacerbated by our somewhat clumsy coding, and the sub-optimal file handling in PalmOS;
3. Our menus are pretty hierarchical, and in particular, once we've chosen a particular patient, we could comfortably ignore the greater part of the database! The caching program permits us to do this 'ignoring'!

Consequently I've decided to write caching routines, which cache files along the lines of the following example:

1. We locate all PROCESSes pertinent to a particular patient, and create a temporary PalmOS 'file' (aka database) called p-PROCESS which is similar to the permanent PROCESS table, but contains only processes relevant to this particular patient.

2. Similarly, we use these identified processes to finger all observations which are relevant, and we strip these out of the EPOCH table, placing them into a temporary p-EPOCH table.

3. When in this *caching scope*, we never access the permanent PROCESS and EPOCH tables. We only access the far smaller temporary tables.

4. Sleight of hand: We will set things up so calls to access relevant tables are via CACHE library whenever we are in the caching scope! Might even consider making this standard, and the CACHE library decides!

5. We might perform similar tricks on all of the subsidiary tables. If we don't cache *all* dependent tables, then our software will detect an apparent loss of relational integrity when it tries to match keys from dependent tables to our cached tables in the course of searches through those dependent tables, but this isn't the major issue it would appear to be, as those searches were doomed to fail anyway!

6. When outside caching scope, table interrogation is as usual (and slow). The 'temporary' tables are deleted on exiting caching scope.

7. Note that because of the clumsiness of PalmOS in finding files, it will probably be wise to eventually 'fix' things so that this library can locate local IDs. We don't do so at present.

8. When no longer needed, we destroy the unwanted temporary files.

1.2 A caching upgrade

Initially, the CACHE instruction on the PDA (we have not yet created something similar in Perl) was very simple. Submit the ID of the *person* of interest, and the library routine does all the rest; submit zero, and all cached files are removed, as is the ability to substitute a p-file for the main database file.

Now we've created a more generic CACHE instruction. If we submit not only the person, but an indication of the chain of databases, we can perform independent caching in several areas. For example, we might say

```
CACHE(PROCESS.Person.1234)
```

. . . the implication being that we identify all PROCESSes referring to a certain Person (with key value 1234).

We might then turn off caching by saying:

```
UNCACHE(PROCESS)
```

With such an approach it would even be possible to maintain many cache threads simultaneously. For example, if we are entering a particular menu while caching is on for a given person, we might cache particular menu components thus:

```
 CACHE(MENUITEMS.miMenu.909)
```

. . . would whittle down MENUITEMS to just those items which reference the given menu. We would later UNCACHE just MENUITEMS to restore the status quo. CACHE(0) might still turn off all.

We would also be able to sequentially cache dependent menus. For example if we have a table EPOCH which depends on PROCESS through its Process key, we say:

```
 CACHE(PROCESS:EPOCH.Process)
```

In order to implement the above, we would need some way of looking up which menus were cached. This lookup should be implemented within the Cache library as a simple linked list rooted in the ‘globals’ within that library. In addition, during the actual caching process, we need some sort of ’recursive’ linking of lists as we pass down the hierarchy (for example PROCESS : EPOCH : NONEVENT).

There is a further problem. If we have a ‘terminal’ table such as NONEVENT (which isn’t referenced in turn by any other table), there’s no point in meticulously storing a linked list of all row keys identified on the offchance that somewhere later we’ll say ‘CACHE(NONEVENT:sometable)’! The wrinkle we introduce is thus:

```
 CACHE(EPOCH:~NONEVENT.Epoch)
```

The tilde signals that we can save space and time by not storing the NON-EVENT row keys. Clearly a tilde in the wrong place (in a table with future dependencies) will cause failure of referential continuity, so this feature should be used with due caution.

1.3 Fine structural detail

For you to understand what we're doing, we must review some structural features of *our* database tables.

1. One of our database tables is represented as a PalmOS 'database' (which we'll call a file).

2. The first record in that file (record 0) contains header information about our database.

3. The format of our database is fully described in the document *PerlPgm.tex*, however we need to know that there is a 32 bit negative number at offset +8 which contains the (negated) number of records in the file.

4. We also must know how to access the column descriptors. You:

   (a) First get the 2 byte number of columns at offset +0xE in record zero: As always, this is big-endian (larger byte leftmost).

   (b) Examine each column in turn. *Offsets* of the column details are located in two byte numbers starting at +10h from the beginning of the record.

   (c) Column details are specified in our column descriptor format, also described in *PerlPgm.tex*. For our purposes the first two bytes contain the relative offset of the column name, and the second two bytes the length of the column name, a maximum of 15 characters. By 'relative offset' we mean relative to the start of the column descriptor.

5. We use knowledge of the above formatting to retrieve the column number of the relevant column within a database, for example we are interested in the column called 'Person' within the PROCESS database, and 'Process' within the EPOCH database.

6. An important feature of our headers is that there is an additional pointer above the last of the column descriptor offsets, which points to one above the very last byte of the header. We can also therefore easily retrieve the length of the header, without asking PalmOS.

7. Similarly, important data can be retrieved from each data row in any of our database tables. At offset +8 we have the (positive) primary key for that line, a 32 bit signed integer.

8. The value for a particular column in any row can be obtained by going to +10h bytes from the start of the row, and then adding 2 bytes per column (The first column is column zero). Get the two-byte value from the resulting offset. The absolute offset (from the start of the row) of the column datum we want is that two-byte value. Its length can be determined from the subsequent offset, and the length of the entire row can be determined by examining the very last offset.

1.4 Includes

Pretty standard includes:

```
#include <SystemMgr.h>
#include <PalmOS.h>

#include "../palmsql3A.h"
  // for MAINCARD etc.
#include "cache.h"
#include "../err/ERRDEBUG.h"
#include "../console/CONSOLE.h"

#define MAXACTUALROWS 2000
// move this to palmsql3A.h
#define MAXTABLES 30
// braces and belt, is this a reasonable limit??
#define MAXIDNODES 500
// hmm. check this.
#define MAXCACHESTORBUF 32+4+MAXACTUALROWS*4
```

1.5 Overview of Main functions

We have a few simple functions:

1. MAKECACHE initially needs to be given a particular Person identified in the PROCESS table. This function will then whip once through the PROCESS table, and while so doing create the p-PROCESS table.[1] At the same time as doing this, MAKECACHE must create an array of process primary keys.

   At future invocations, MAKECACHE must then similarly abstract relevant rows from EPOCH into p-EPOCH, which will shrink dramatically. It uses the array of process primary keys to determine which EPOCH rows should be retained in p-EPOCH.

[1]We use the braces-and-belt approach of deleting all old p-tables if they are detected.

2. We might stop there, but it is tempting to use similar mechanisms on all files dependent on EPOCH (etc). Note that if we do this, we must similarly retain an array of epoch primary keys, and retain rows in other p-files based on these epoch primary keys (and even if necessary, on down the line).

3. Note that (at present) only SQL *queries* will use the p-files. However, every time we alter a non-p file, we will need to update or reconstruct the relevant p-file (that is, if it exists).

4. ISCACHING returns a value of 1 if caching is occurring, 0 if not, and under zero if caching is disabled.

5. KILLCACHE deletes *all* p-files.

6. CACHEFINDTABLE is simplicity itself. When the SQL library invokes its CreateTableList function, it invokes CACHEFINDTABLE which if caching is off, returns an unchanged table name, otherwise altering the name to p-TABLENAME. CreateTableList must supply the name in a buffer with space for two extra bytes so that the p- prefix can be inserted! Subsequent searches then simply use the pared down form of the table!

7. CACHEINSERT meets the problem of a new row being inserted into e.g. PROCESS. It simply inserts the same row at an appropriate place in p-PROCESS.

8. CACHEUPDATE is similar to CACHEINSERT, finding and *replacing* the relevant row in the cache file whenever a row is updated in the source.

9. DISABLECACHE disables all caching.

2 Basic functions

The following are fairly standard startup functions.

```
Err start (UInt16 refnum, SysLibTblEntryPtr entryP)
{
  extern void *jmptable ();
  entryP->dispatchTblP = (void *) jmptable;
  entryP->globalsP = NULL;
  return 0;
}
```

During caching, we will store temporary data in a clumsy linked list (rather than having a clumsy array) thus:

```
typedef struct {
  Int32 key;
  void * next;
  } cacheNode;
```

The idea is that as we select out 32-bit primary keys, we will need to store them in a list for future reference. See later usage. Next, a simple data structure used for storing the name of a cached table. The convoluted typedef is ‘just because’ of the design of C. Otherwise we have trouble with self-referential pointers.

```
typedef struct TableNode TableNode;

struct TableNode {
  Char * tname;
  Int16 nlen;
  cacheNode * data;
  TableNode * next;
  };
```

In the above structure, we store the name of the data table (and the length of the name), as well as a cacheNode pointer to all the selected 32-bit keys, which will be stored in the reverse order in which they are encountered.

Here are the globals:

```
typedef struct {
  UInt16 CONSOLE;
  UInt16 ERRLIB;
  UInt16 DEBUGFLAGS;
  UInt16 refcount;
  Int16 ENABLED;
  TableNode * TABLELIST;
  } CacheLib_globals;
  // include count in expectation of multiple openings!
```

We root our *most recent* TableNode in the globals, and it in turn points to the next most recent, and so on down the line.

```
Err CACHEOpen (UInt16 refnum)
 {
  SysLibTblEntryPtr entryP = SysLibTblEntry (refnum);
  CacheLib_globals *gl = entryP->globalsP;
  if (!gl)
     {
       gl = entryP->globalsP = MemPtrNew (sizeof (CacheLib_globals));
       MemPtrSetOwner (gl, 0); // note use of sys fxs here, above
       gl->CONSOLE = 0;
       gl->ERRLIB = 0;
       gl->DEBUGFLAGS = 0;
       gl->refcount = 0;
       gl->ENABLED = 0; // off at start.
       gl->TABLELIST = 0;
    }
  gl->refcount ++;
  return 0;
 }
```

We have to allocate memory for our globals using MemPtrNew; ideally we should wrap the system calls as usual (Here using `C_` rather than `W_`).

```
Err CACHEClose (UInt16 refnum, UInt16 *numappsP)
{
  SysLibTblEntryPtr entryP = SysLibTblEntry (refnum);
  CacheLib_globals *gl = entryP->globalsP;

  if (!gl) {
    return 1; // not open
    }

  // clean up:
  *numappsP = --gl->refcount;
  if (*numappsP == 0) {

    // here would free up open files, delete temp files ... ?
    MemChunkFree (entryP->globalsP);
    entryP->globalsP = NULL;
    }
  return 0;
}

Err nothing (UInt16 refnum) {
  return 0;
  }
```

2.1 Some wrappers

The following were lifted from *Sql3Lib.tex*:

```
Int16 c_MemPtrFree (MemPtr p)
{ Int16 ok;
  if (p == 0)
   { return 0;
   };
  ok = MemPtrFree (p);
  if (!ok) { return 1; };
  return 0; // clumsy
}


Int16 c_MemPtrUnlock (MemPtr p)
{Int16 ok;
  if (p == 0)
   { return 0;
   };
  ok = MemPtrUnlock (p);
  if (!ok) { return 1; };
  return 0;
}


MemHandle c_MemHandleNew (UInt32 size)
{ if (! size)
   { return 0;
   };
  return MemHandleNew (size);
}


MemPtr c_MemHandleLock (MemHandle h)
{ if (! h )
   { return 0;
   };
  return MemHandleLock (h);
}


Int16 c_MemHandleUnlock (MemHandle h)
{ Int16 ok;
  if (h == 0)
   { return 0;
   };
  ok = MemHandleUnlock (h);
  if (!ok) { return 1; };
  return 0;
}
```

2.2 A few simple utilities

2.2.1 Copy

xCopy is straightforward (but clumsy):

```
Int16 xCopy (Char * dest, Char * xsrc, Int16 cnt)
{ if (! dest)
     { return 0;
     };
  if (! xsrc)
     { return 0;
     };
  while (cnt > 0)
    { *dest++ = *xsrc++;
      cnt --;
    };
  return 1;
}
```

2.2.2 Same

The following simple comparison returns just yes (1 = identical strings), or zero. If the lengths differ, the strings cannot be the same, of course.

```
Int16 c_Same (Char * p0, Int16 p0len, Char * p1, Int16 p1len)
{
  if (p0len != p1len)
     { return 0;
     };

  while (p0len > 0)
    { if (*p0 != *p1)
         { return 0; //fail
         };
      p0len --;
      p0 ++;
      p1 ++; // clumsy.
    };
  return 1; // identical strings
}
```

2.3 Debugging and error handling

2.3.1 Some memory management

The following routines are minor modifications of memory handlers specified in *CProgMain.tex*.

```
Char * xNew (Int16 memsize)
{ MemHandle memH=0; // clumsy.
  MemPtr    memP=0;

  memH = c_MemHandleNew(memsize);
  if (! memH)
     { return 0;
     };
  memP = c_MemHandleLock(memH);
  if (! memP)
     { return 0;
     };
  return ((Char *) memP);
}

Int16 Delete (MemPtr memP)
{
  if (! memP)
     { return 1; // ugly
     };
  if (! c_MemPtrUnlock (memP))
     { return 0; // fail
     };
  if (! c_MemPtrFree (memP))
     { return 0;
     };
  return 1; // success
}
```

2.3.2 Write text to console

Now, the routine to actually write text to the console:

```
void ConTx_c(UInt16 refnum, Char * txt, Int16 txlen,
                      UInt16 bugflag)
{
  UInt16 CONSOLE;
  SysLibTblEntryPtr entryP;
  CacheLib_globals *gl;
  UInt16 dbg;

  entryP = SysLibTblEntry (refnum);
  gl = entryP->globalsP;
  CONSOLE = gl->CONSOLE;
  dbg = gl->DEBUGFLAGS;

  if (bugflag) // okay, might use &&
     { if (! (dbg & bugflag))
```

```
          { return;
          };
      };
  if (! CONSOLE)
      { return;  // fail
      };
  // now for some sanity checks:
  if (txlen < 1)
      { ConWrite(CONSOLE, "<Z>", 3);
        return;
      };

  if (! txt)
      { ConWrite(CONSOLE, "<P>", 3);
        return;
      };

  // if leading ? ie error write, newline:
  if (*txt == '?')
      { ConWrite (CONSOLE, "\n", 1);
      };
  // finally, write:

   ConWrite(CONSOLE, txt, txlen);
}
```

In the above, if `bugflag` is zero, then we 'always' write to the console — setting this value to zero forces a write, regardless of the flags. This convention is to allow forced writing of short error messages to the console, and should only be used for this purpose. We define the constant ZERO to allow us to identify such statements.

```
Int16 c_StrLen (Char * txt)
{
  if (! txt) { return 0; };
  return StrLen(txt);
};
```

Here's a version which uses ASCIIZ strings:

```
void ConAsc_c(UInt16 refnum, Char * txt, UInt16 bugflag)
{
  Int16 txlen;
  txlen = c_StrLen(txt);
  ConTx_c(refnum, txt, txlen, bugflag);
}
```

2.3.3 Write Integer to console

```
Int16 c_StrIToA (Char *s, Int16 slen, Int32 i)
{  if (slen < 11)
      { return 0;
      };
   StrIToA (s, i);   // can this fail?
   return ((Int16) c_StrLen(s));
}
```

ConI_c is similar to *ConTx_c*.

```
void ConI_c (UInt16 refnum, Int32 i, UInt16 bugflag)
{ UInt16 CONSOLE;
  SysLibTblEntryPtr entryP;
  CacheLib_globals *gl;
  UInt16 dbg;
  Int16 ilen;
  Char * CRUTCH2;

  entryP = SysLibTblEntry (refnum);
  gl = entryP->globalsP;
  CONSOLE = gl->CONSOLE;
  dbg = gl->DEBUGFLAGS;

  if (bugflag) // okay, might use &&
     { if (! (dbg & bugflag))
          { return;
          };
     };

  CRUTCH2 = xNew(maxStrIToALen+1);
  ilen = c_StrIToA (CRUTCH2, maxStrIToALen+1, i); // sys fx.
  ConWrite (CONSOLE, CRUTCH2, ilen);
  Delete(CRUTCH2);
}
```

2.3.4 Pass handles to library

Rudimentary at present, modelled on *Sql3Lib.tex*.

```
Int16 PassCacheBug (UInt16 refnum, UInt16 bugs, UInt16 errlib,
                    UInt16 console)
{
  SysLibTblEntryPtr entryP = SysLibTblEntry (refnum);
  CacheLib_globals *gl = entryP->globalsP;
  if (!gl)
  {  return -1; // fail (value < 0 = failure)
  };
  gl->DEBUGFLAGS = bugs;
  gl->ERRLIB = errlib;
  gl->CONSOLE = console;
  gl->ENABLED = 1; // permit caching
  return 1; // hmm. signal ok ?
}
```

2.3.5 Advance routine

How many characters is it to *one after* the next occurrence of character **c**? Return 1 if the character is at the start of the line, 0 on failure! In other words, return the length of the string up to and including that character. Fail (return 0) if the character is not found by the time **limit** characters have been examined.

```
Int16 c_Advance (Char * myptr, Int16 limit, Char c)
{
  Int16 howfar = 0;
  while (limit > 0)
    { howfar ++;
      if (* myptr == c)
         { return howfar;
         };
      limit --;
      myptr ++; // clumsy
    };
  return 0; // fail
};
```

2.3.6 Read an integer

The following routine is very primitive, and will only read positive integers, failing (with -1) on negative ones, at present. Snitched from *Sql3Lib.tex*.

```
Int32 c_Ascii2I32 (Char * P, Int16 ilen)
{ Int32 i;
  Char c;

  i = 0;
  while (ilen > 0)
    { c = *P++;
      ilen --;
      if ((c < ’0’) || (c > ’9’)) // test me with char = e.g. 0xF0
         { return -1;             // fail
         };
      c -= ’0’;
      i = (i*10) + c;
    };
  return i;
}
```

2.4 Create a database

Given a database name and the length of that name, create a new database. Fail if the database already exists. We assume the database is NUL-terminated (ugh).

```
Int16 c_DmCreateDatabase (Char *nameP)
{
  Int16 fail;
  if (! nameP)
   { return 0;
   };
  fail = DmCreateDatabase (MAINCARD, nameP, DBCREATOR,
                           DBTYPE, false);
  // MAINCARD etc are included in palmsql3A.h
  if (!fail) { return 1; };      // success
//  ERRmsg(ErFailMakeDb);
  return 0;
}
```

2.5 Find, open database

The following finds the database, not modifying the search name, and assumes an ASCIIZ string was supplied (!)

```
LocalID c_DmFindDatabase (Char* nameP)
{
  LocalID lid;

  lid = DmFindDatabase(MAINCARD, nameP); // does database exist?

  return (lid);
}
```

The following was lifted from *IdxLib.tex*

```
DmOpenRef c_DmOpenDatabase (LocalID dbID, UInt16 mode)
{
  return DmOpenDatabase (MAINCARD, dbID, mode);
}
```

We also need to be able to close our databases:

```
Int16 c_DmCloseDatabase (DmOpenRef dbP)
{ Int16 ok;
  if (dbP == 0)
   { return 0;
   };
  ok = DmCloseDatabase (dbP);
  if (!ok) { return 1; };
  return 0;
}
```

DmCloseDatabase returns zero if *no* error occurred, a nonzero value if an error did occur. We flip these around so s_DmCloseDatabase returns zero if an error occurred.

In this library we also can delete (temporary) databases, given the (ugh) local ID.

```
Int16 c_DmDeleteDatabase (LocalID dbID)
{ Int16 ok;
  if (! dbID)
   { return 0;
   };
  ok = DmDeleteDatabase (MAINCARD, dbID);
  // 0 signals NO error so:
  if (!ok) { return 1; };
  return 0;
}
```

2.6 More database functions

2.6.1 Write to a record

Write data from a string to a record within a database at the specified offset in the record. We have several sanity checks. We permit writing of zero bytes in which case we simply ‘succeed’!

```
Int16 c_DmWrite (void *recordP, UInt32 offset,
                        const void *srcP, UInt32 bytes)
{ Int16 ok;
  if (recordP == 0)
   { // ERRmsg(ErWriteNulRec);
     return -1;
   };
  if (srcP == 0)
   { // ERRmsg(ErWriteNulSrc);
     return -2;
   };
  if (bytes == 0)
   { return 0; // permit 0 byte write!
   };

  // what is time penalty for the following check? (smallish)
  ok = DmWriteCheck(recordP, offset, bytes);
  if (ok != errNone)
     { // ERRmsg(ErFailRecCheck);
       return -3; // fail
     };
  ok = DmWrite (recordP, offset, srcP, bytes);
  if (!ok) { return ((Int16) bytes); };  // bytes written
   // ERRmsg(ErFailRecWrite);
  return -4;
}
```

Comments as for the rtn with the same name in *CProgMain.tex*.

2.6.2 Making a new record

The PalmOs function DmNewRecord uses the value stored in *atP* as an index, creating a new record of the given size as the row specified by the index. The first row is row zero.

```
MemHandle c_DmNewRecord (DmOpenRef dbP, UInt16 *atP,
                                UInt32 size)
{
  return DmNewRecord (dbP, atP, size);
}
```

If we try to make a new record above the topmost record, specifying an index which is not the immediate record above the topmost one, we're in for a surprise. PalmOS does *not* insert dummy new records, merely making the new one at the very top after the current topmost record, and returning the index of this record (row) in atP.

2.6.3 Releasing a database record

Release a record, given the database and the index of the record. Should be used after DmGetRecord, not needed for DmQueryRecord. The dirty flag bit is set in the PalmOs record if the final parameter of DmReleaseRecord is 1. For our caching, we always keep this reset.

```
Int16 c_DmReleaseRecord (DmOpenRef dbP, UInt16 index)
{ Int16 ok;
```

```
  if (dbP == 0)
   { return -1;
   };
  ok = DmReleaseRecord (dbP, index, 0); // 0-> NOT dirty
  if (!ok) { return 1; };
  return 0; // clumsy
}
```

2.6.4 Write a new record

Next, *our* actual record creation. We create the record and then write the given string to the record. After writing, we unlock and release the record.

```
Int16 MakeFileRecord (UInt16 refnum,
                      DmOpenRef myDB, UInt16 idx,
                      const Char* recdata, Int16 datalen)
{ MemHandle mynewrec;
  void * recPtr;
  UInt16* pIdx;
  Int16 desiredIdx = idx;
  pIdx = &idx; //point to index

  mynewrec = c_DmNewRecord(myDB, pIdx, datalen);
  if (! mynewrec)
     { // ERRmsg(ErFailNewRecord);
       return 0; //fail
     };

  // WARN if idx is too great:
  if (idx != desiredIdx)
     { // desired is greater than last idx record??
       ConTx_c( refnum, "\n?IDX++:", 8, 0);
       ConI_c (refnum, idx, 0);
       ConTx_c( refnum, "/", 1, 0);
       ConI_c (refnum, desiredIdx, 0);
     };

  recPtr = c_MemHandleLock(mynewrec); // could fail?
  if (c_DmWrite(recPtr, 0, recdata, datalen) != datalen)
     { // ERRmsg(ErFailWriteRecord);
       return 0;
     };
  if (! c_MemHandleUnlock(mynewrec))
       // do NOT use recPtr!
     { // ERRmsg(ErFailUnlockRecord);
       return 0;
     };
  if (! c_DmReleaseRecord(myDB, idx))
```

```
     { // ERRmsg(ErFailReleaseRecord);
       return 0;
     };
  return 1;  // success.
}
```

2.6.5 Open Record for reading

DmQueryRecord opens a record for *reading* only.

```
MemHandle c_DmQueryRecord (DmOpenRef dbP, UInt16 index)
{
  return DmQueryRecord (dbP, index);
}
```

2.6.6 Open Record, can write

```
MemHandle c_DmGetRecord (DmOpenRef dbP, UInt16 index)
{
  return DmGetRecord (dbP, index);
}
```

2.6.7 Resize a record

Given the index of a record, resize it. The handle returned is that of the resized record, or NULL on failure.

```
MemHandle c_DmResizeRecord (DmOpenRef dbP, UInt16 index,
                                   UInt32 newSize)
{
  return DmResizeRecord (dbP, index, newSize);
}
```

2.6.8 Find insertion position

Determine where a record should be inserted:

```
UInt16 c_DmFindSortPosition (DmOpenRef dbP, void *newRecord,
                             SortRecordInfoPtr newRecordInfo,
                             DmComparF *compar, Int16 other)
{
  if (dbP == 0)
   { return 0;
   };
  if (newRecord == 0)
   { return 0;
   };
  if (compar == 0)
   { return 0;
   };
  return DmFindSortPosition (dbP, newRecord, newRecordInfo,
                             compar, other);
}
```

3 Important subsidiary functions

3.1 Node handling

3.1.1 NewIdNode

Given a 32 bit ID, create a new node, populate it, and return a pointer to the node. Above we described the cacheNode structure, used for temporarily storing 32 bit IDs.

```
cacheNode* NewIdNode( UInt16 refnum,
                 Int32 id)
{
  cacheNode * n;
  n = (cacheNode *) xNew( sizeof(cacheNode));
  n->key = id;
  n->next = 0; // by default, terminated.
  return n;
}
```

3.1.2 FindIdNode

Given a 32 bit ID, and the root of a node list, traverse all nodes until end (0) or find target ID (1).

```
Int16 FindIdNode ( UInt16 refnum, cacheNode * root,
                 Int32 id)
{
  Int16 braces=0;

//  +OPTIONAL
//     ConTx_c( refnum, "\n@", 2, 0);
//     ConI_c (refnum, id, 0);
//     ConTx_c( refnum, ":", 1, 0);
//  -OPTIONAL

  while (   root
         && (braces < MAXIDNODES)
        )
    { braces ++;
      if (root->key == id)
         { return 1;
         };
//     +OPTIONAL
//       ConI_c (refnum, root->key, 0);
//       ConTx_c( refnum, " ", 1, 0);
//     -OPTIONAL
```

```
      root = (cacheNode *) root->next;
    };
  if (braces >= MAXIDNODES)
     { return -10;
     };
  return 0;
}
```

3.1.3 KillAllIdNodes

Given the root of a cachenode list, delete all nodes. Return the number of nodes deleted, or -1 if the cacheNode pointer submitted is null.

```
Int16 KillAllIdNodes ( UInt16 refnum, cacheNode * root)
{
  Int16 hits=0;
  Int16 braces = 0;

  cacheNode * nxt;
  if (! root)
     { return -1; // not found
     };

  while (  (root)
         &&(braces < MAXIDNODES)
        )
    { nxt = (cacheNode *) root->next;
      Delete ((Char *) root); // might check success?
      root = nxt;
      hits ++;
    };
  if (braces >= MAXIDNODES)
     { return -10;
     };

  return hits;
}
```

3.2 Handle caching flags

3.2.1 Turn on caching

```
Int16 TurnOnCaching (UInt16 refnum)
{
  SysLibTblEntryPtr entryP = SysLibTblEntry (refnum);
  CacheLib_globals *gl = entryP->globalsP;
  if (!gl)
  {  return -1; // fail
  };
  // does not test if on already:
  gl->ENABLED = 1;
  return 1; // ok.
}
```

3.2.2 Turn OFF caching

```
Int16 TurnOffCaching (UInt16 refnum)
{
  SysLibTblEntryPtr entryP = SysLibTblEntry (refnum);
  CacheLib_globals *gl = entryP->globalsP;
  if (!gl)
  {  return -1;
  };
  // (doesn't test if off, might do so.)
  gl->ENABLED = 0;
  return 1; // ok.
}
```

3.2.3 Get caching status

Returns 1 if on, 0 if off, -1 if error.

```
Int16 GetCaching (UInt16 refnum)
{
  SysLibTblEntryPtr entryP = SysLibTblEntry (refnum);
  CacheLib_globals *gl = entryP->globalsP;
  if (!gl)
     { return -100;
     };
  return (gl->ENABLED);
}
```

3.3 Database name handling

Here we explore functions which allow us to:

1. Determine whether a particular database is indexed, and if indexed, return the TableNode pointer to the database reference (from which we can obtain the cacheNode root for this table).
2. Store the name of a newly indexed database in the CacheLib_globals structure;

3. Remove an indexed database from the list rooted in the CacheLib_globals structure. [? what about children];

4. Delete (kill) a TableNode structure, including the name and cacheNode data list.

3.3.1 Find database by name

```
TableNode * FindDatabaseByName (UInt16 refnum, Char * name, Int16 nlen)
{
  TableNode * tn;
  SysLibTblEntryPtr entryP = SysLibTblEntry (refnum);
  CacheLib_globals *gl = entryP->globalsP;
  Int16 braces=0; // max table depth

  if (!gl)
     { return 0; // fail
     };
  if (! gl->ENABLED)
     { return 0; // fail (might signal error?)
     };
  if (! name)
     { return 0; // nonsense
     };

  tn = gl->TABLELIST;
  while (  (tn)
         &&(braces < MAXTABLES)
        )
    { braces ++;
      if (tn->nlen == nlen)
         { if ( c_Same(tn->tname, tn->nlen, name, nlen) )
              { return (tn);
              };
           //  +OPTIONAL
           //   ConTx_c( refnum, "\n?db:", 5, 0);
           //   ConTx_c( refnum, tn->tname, tn->nlen, 0);
           //   ConTx_c( refnum, "(", 1, 0);
           //   ConI_c (refnum, nlen, 0);
           //   ConTx_c( refnum, ")", 1, 0);
           //  -OPTIONAL
         };
      tn = tn->next;
    };
  if (braces >= MAXTABLES)
     {
       +OPTIONAL
        ConTx_c( refnum, "\n*ERROR:Loop FindDb*", 20, 0);
       -OPTIONAL
     };
  return 0; // not found.
}
```

3.3.2 Store database name

Given a name, create a new TableNode and return a pointer to it, or fail, returning null. The name should *not* have a ‘p-’ prefix.

```
TableNode * StoreDatabaseName (UInt16 refnum, Char * tname, Int16 nlen)
{
  SysLibTblEntryPtr entryP;
  CacheLib_globals *gl;
  TableNode * tn;

  if ( FindDatabaseByName (refnum, tname, nlen) )
     { return 0; // fail
     };  // MAKE SURE not already registered!!

  entryP = SysLibTblEntry (refnum); // clumsy.
  gl = entryP->globalsP;

  if (!gl)
     { return 0;
     };
  if (! gl->ENABLED)
     { return 0;
     };
  tn = (TableNode *) xNew( sizeof(TableNode) );
  if (! tn)
     { return 0; // fail
     };

  tn->tname = xNew(nlen);
  tn->nlen = nlen;
  xCopy(tn->tname, tname, nlen);
  tn->data = 0; // nb.

  tn->next = gl->TABLELIST; // can be null
  gl->TABLELIST = tn; // store
  return tn; // success.
}
```

3.3.3 Remove database node

We excise the TableNode from the root, returning a pointer to the excised node!

```
TableNode * RemoveDatabaseNode (UInt16 refnum, Char * name, Int16 nlen)
{
  TableNode * tn;
  TableNode * prev;
  SysLibTblEntryPtr entryP = SysLibTblEntry (refnum);
  CacheLib_globals *gl = entryP->globalsP;
  Int16 braces = 0;

  if (!gl)
     { return 0; // fail
     };
  if (! gl->ENABLED)
     { return 0; // fail (? signal this)
     };
  if (! name)
     {
       ConTx_c( refnum, "uncache: no name?", 17, 0);
       return 0; // nonsense
     };

      // debug:
      //  ConTx_c(refnum, "\nN:", 3, 0);
      //  ConTx_c(refnum, name, nlen, 0); // jvs
      // end debug.

  prev = 0;
  tn = gl->TABLELIST;
  while (  (tn)
         &&(braces < MAXTABLES)
        )
    { braces ++;
      // debug:
      //  ConTx_c(refnum, "\n?n:", 4, 0);
      //  ConTx_c(refnum, tn->tname, tn->nlen, 0); // jvs
      // end debug.
      if (tn->nlen == nlen)
         { if ( c_Same(tn->tname, tn->nlen, name, nlen) )
              { if (! prev) // if root!
                   { gl->TABLELIST = tn->next;
                   } else
                   { prev->next = tn->next;
                   };
                return tn; // SUCCESS: return excised node.
              };
         };
      prev = tn; // keep reference!
      tn = tn->next; // fix: 19/8/2007 [twit]
    };
  if (braces >= MAXTABLES)
     {
       +OPTIONAL
        ConTx_c( refnum, "\n*ERROR:Loop RemDb*", 19, 0);
       -OPTIONAL
     };
  ConTx_c(refnum, "\nbad uncache:", 13, 0);
  ConTx_c(refnum, name, nlen, 0);
  return 0; // failed.
}
```

3.3.4 Kill p-table

Given the name of a p-table, without the suffix, locate and delete the table.

```
Int16 Killp_table (UInt16 refnum, Char * tname, Int16 nlen)
{
  Char * p_name;
  LocalID lid;

  if ((! tname) || (nlen < 1))
     { return -1; // error
     };

  p_name = xNew(nlen+3);
  if (! p_name)
     { return -2;
     };
  xCopy(p_name+2, tname, nlen);
  xCopy (p_name, "p-", 2);
  * (p_name+nlen+2) = 0x0; // asciiz.

  lid = c_DmFindDatabase (p_name); // slooow.
  Delete(p_name);

  if (! c_DmDeleteDatabase(lid))
     { return 0; // blaagh.
     };
  return 1; // success.
}
```

3.3.5 Kill TableNode

given a node, kill the associated cacheNode, then the node itself. Also delete the associated p-table.

```
Int16 KillTableNode (UInt16 refnum, TableNode * tn, Int16 killp)
```

```
{
  Int16 ok = 1;
  if (!tn)
     { return -1;
     };
  if (KillAllIdNodes(refnum, tn->data) < 0)
     { ok = 0;
     }; // clumsy

  if (killp)
     { if ( Killp_table (refnum, tn->tname, tn->nlen) < 1 )
          { ok = 0;
          };
     }; // clumsy but explicit

  if (! Delete ((Char*)(tn->tname)) )
     { ok = 0;
     };

  if (! Delete ((Char *)tn) )
     { ok = 0;
     };

  return ok; // 0 or less means failed, 1 = success.
}
```

3.3.6 Find and delete table node

We first chop the relevant node out of the linked list of active table nodes using RemoveDatabaseNode, and then delete the node in its entirety.

```
Int16 ObliterateTableNode (UInt16 refnum, Char * tname, Int16 nlen)
{
  TableNode * tn;
  tn = RemoveDatabaseNode (refnum, tname, nlen);
  if (! tn)
     { return -1;
     };
  return ( KillTableNode (refnum, tn, 1) ); // kill p-file too!
}
```

3.3.7 Kill *all* table nodes

Walk through linked list of table nodes, deleting each one, and the associated p-file.

```
Int16 KillAllTableNodes (UInt16 refnum, Int16 killp)
```

```
{
  TableNode * tn;
  TableNode * nxt;
  SysLibTblEntryPtr entryP = SysLibTblEntry (refnum);
  CacheLib_globals *gl = entryP->globalsP;
  Int16 braces=0;

  if (!gl)
     { return 0; // fail
     };
  if (! gl->ENABLED)
     { return 2; // already killed (!?)
     };

  tn = gl->TABLELIST;
  gl->TABLELIST = 0; // detach.
  while (  (tn)
         &&(braces < MAXTABLES)
        )
    { braces ++;
      nxt = tn->next;
      KillTableNode (refnum, tn, killp);
      tn = nxt;
    };
  if (braces >= MAXTABLES)
     { return -10; // serious error
     };
  return 1; //ok
}
```

3.4 Retaining the cache on exit

Here we'll define functions to write the complete current state of the cache to a file, and then reload that state on re-entering the cache library. This is a significant task. All data are written to CACHESTORE.SQ3 on the PDA.

Once the state has successfully been stored, the entire caching system is turned off, and the associated nodes are deleted!

3.4.1 Storing the cache state

We must:

1. First store the (linked list) of TableNode nodes, one per record in CACHESTORE.SQ3;

2. With *each* TableNode, the associated cacheNode linked list is stored too!

3. *Ensure* that if new data are loaded from the PC, the .SQ3 files are disabled or deleted (Most secure is to write dummy, zeroed files from the PC as a routine part of transferring files)!

We swop things around a little in storing — for each TableNode, first we store whole record size, then the length of the table name (tname), then the ASCII string for that name, and then a 32 bit value for each cachenode. If the initial area containing the table name has a length not divisible by 4, then we pad up to this length (0–3 bytes).

Note that the very first 2 bytes of CACHESTORE.SQ3 record 0 will ultimately contain the *actual* number of TableNodes. If this value is zero, then the whole of the file is invalid. [EXPLORE THIS CASE]! The *second* two bytes will contain the maximum number of TableNodes.

We will store one TableNode with all associated data. The first TableNode will go to record 1, the next to record 2 and so on. Record 0 will be reserved for meta-information about the TableNodes, for now simply the number of them.

If the ENABLED flag is reset, we do none of the above.

```
Int16 CACHESTORESTATE (UInt16 refnum)
{ TableNode * tn;
  cacheNode * cn;
  SysLibTblEntryPtr entryP = SysLibTblEntry (refnum);
  CacheLib_globals *gl = entryP->globalsP;

  LocalID   wrid; // usual ugly stuff..
  DmOpenRef s_openref;
  MemHandle hmem;
  Int32 stuff = 0; // clumsy
  Int16 tablecount = 1; // NB=1.
  Int16 oldtablecount = 0; // will update!

  Char * BUFFER;
  Char * pBUF;
  Int16 bufsize;
  Int16 ok;
  Char * pRes;
  Int16 templen; // hack

  // return(-(refnum)); /// debug

  if (!gl)
     { return 0; // fail
     };
  if (! gl->ENABLED)
     { return -1; // fail
```

```
   };

// here open file:
wrid = c_DmFindDatabase ("CACHESTORE.SQ3");
if (! wrid)
   {  if (! c_DmCreateDatabase ("CACHESTORE.SQ3"))
        { return -2; // just fail ??
        };
      wrid = c_DmFindDatabase ("CACHESTORE.SQ3");
      s_openref = c_DmOpenDatabase (wrid, dmModeReadWrite);
      stuff = 0x00000001; // nb ENDIAN storage below!!!
      if (! MakeFileRecord (refnum, s_openref, 0, (Char *)&stuff, 4)) // [ENDIAN
           // at offset +0 = 2 byte ACTUAL record count
           // at offset +2 = 2 byte MAXIMUM record count (for now)
         { return -3; // make record #zero
         };
   } else
   { s_openref = c_DmOpenDatabase (wrid, dmModeReadWrite);
   };
if (! s_openref)
   { return -7;
   };

// get number of records, _including_ record 0:
hmem = c_DmQueryRecord(s_openref, 0); // must exist
if (! hmem)
   { return -8;
   };
pRes = (Char *) c_MemHandleLock(hmem);
oldtablecount = * ( (Int16*)(pRes+2)); // read 'old' MAXIMUM count! [big-ENDIAN
// later may rewrite new count (if greater than old).
// [potential flaw here if file corrupted ?!][if zero might rewrite file]
if (! c_MemHandleUnlock(hmem))
   { return -4;
   };

// create buffer:
BUFFER = xNew(MAXCACHESTORBUF);

// walk through tables:
tn = gl->TABLELIST;
while (tn)
  { // first write data to buffer area:
    *((Int16*)(BUFFER+2)) = tn->nlen; // store length[ENDIAN]
    xCopy ((BUFFER+4), tn->tname, tn->nlen); // store name
    templen = 4 + 3 + tn->nlen;
    templen &= -4; // ie. 0xFF..FC
    pBUF = BUFFER+templen;
    // Above forces integral boundary divisible by 4. [NB]

    cn = tn->data;
    while (cn)
      { stuff = cn->key;
        *((Int32*)(pBUF))=stuff; // clumsy
        pBUF += 4;
        cn = cn->next;
      };
    bufsize = (Int16)(pBUF - BUFFER);
    *((Int16*)(BUFFER)) = bufsize; // at start!
    // might also have b&b limiting count (dud cn->next??)

    // NEXT WRITE RECORD. Must resize!
    // Start = record 1.
    if (tablecount >= oldtablecount)
       { oldtablecount ++;
         MakeFileRecord (refnum, s_openref, tablecount, BUFFER, bufsize);
          // might check for failure!
       } else
       { hmem = c_DmGetRecord(s_openref, tablecount); // hmm 2007-12-09
         hmem = c_DmResizeRecord(s_openref, tablecount, bufsize);
         if (!hmem)
            { Delete(BUFFER);
              c_DmCloseDatabase(s_openref);
              return -9; // aagh
            };
         pRes = (Char *) c_MemHandleLock(hmem);
         ok = c_DmWrite( pRes, 0, BUFFER, bufsize);
         c_MemHandleUnlock(hmem);
         c_DmReleaseRecord(s_openref, tablecount); // vs DmGetRecord?
         /// cf: http://www.mail-archive.com/palm-dev-forum@3com.com/msg10099.ht
         if (ok < bufsize) // write failed
            { Delete(BUFFER);
              c_DmCloseDatabase(s_openref);
              return -10; // $jvs$
            };
       };
    // next row..
    tablecount ++; // bump
    tn = tn->next;
  };
Delete(BUFFER);

hmem = c_DmGetRecord(s_openref, 0); // update meta-data in record 0.
pRes = (Char *) c_MemHandleLock(hmem);
c_DmWrite( pRes, 0, (Char *)&tablecount, 2); // write new rec count!
c_DmWrite( pRes, 2, (Char *)&oldtablecount, 2); // old count. [ugly]
```

```
  c_MemHandleUnlock(hmem);       // might test
  c_DmReleaseRecord(s_openref, 0); // might test

  if (! c_DmCloseDatabase(s_openref))
     { return -6;
     };

  KillAllTableNodes (refnum, 0); // retain p-files
  TurnOffCaching (refnum); // ugly
  return 1; // success
}
```

3.4.2 Restoring the cache

We perform the opposite of the above, restoring both the TableNode linked list and the cacheNode list. We *must* also remember to set gl->ENABLED, or the library will fail to see the cache.

There is a bit of a problem here. If we 'cached' things but the cache was not active, (there is nothing to do) then current 'failure to cache' is actually not an error state and shouldn't be treated as such! We therefore distinguish between returning zero ('No cache restore, but this is OK') and values under zero ('Cache failure error')!

```
Int16 CACHESTATERESTORE (UInt16 refnum)
{
  TableNode * tnode;
  cacheNode * cn;
  SysLibTblEntryPtr entryP = SysLibTblEntry (refnum);
  CacheLib_globals *gl = entryP->globalsP;

  LocalID   wrid; // usual ugly stuff..
  DmOpenRef storref;
  MemHandle hmem;
  Int32 stuff = 0; // clumsy
  Int16 tablecount;
  Int16 rec;
  Int16 recsize;
  Int16 namlen;
  Char * pRes;

  if (!gl)
     { return -1; // fail
     };
 // if (gl->ENABLED)
 //    { return -2; // fail IF ALREADY *ENABLED*
 //    };  // later modify to allow this 'feature' ?!
  // here open file:
  wrid = c_DmFindDatabase ("CACHESTORE.SQ3");
  if (! wrid)
     { return (0); // simple failure
     };
  storref = c_DmOpenDatabase (wrid, dmModeReadOnly);  // [???????? rewrite to 0]
  if (! storref)
     { return -3;
     };

  // get number of records, _including_ record 0:
  hmem = c_DmGetRecord(storref, 0); // should exist
  if (! hmem)
     { return -4;
     };
  pRes = (Char *) c_MemHandleLock(hmem);
  tablecount = * ( (Int16*)(pRes)); // read count [ENDIAN]
    // do NOT read count at offset +2 as this is MAX count.
  c_MemHandleUnlock(hmem);
  c_DmReleaseRecord(storref, 0);
  if (tablecount < 1)
     { c_DmCloseDatabase(storref);
       return (0); // simple failure
     };

  // next, re-create table nodes:
  rec = 1;
  while (rec < tablecount)
    { // read table name, and create a new TableNode
        hmem = c_DmQueryRecord(storref, rec);
          if (! hmem)
             { c_DmCloseDatabase(storref);
               return -(100+rec); // fail
             };
        pRes = (Char *) c_MemHandleLock(hmem);
        recsize = *((Int16 *)(pRes)); // [ENDIAN]
         // [might confirm recsize is divisible by 4]
        namlen = *((Int16 *)(pRes+2)); // [ENDIAN]
        tnode = StoreDatabaseName(refnum, pRes+4, namlen);
         // this automatically updates gl->TABLELIST

        pRes += recsize; // move to END!
        namlen += 7; // 4 for lengths, 3 for integral boundary
        namlen &= -4; // i.e. 0xFF..FC
        recsize -= namlen; // find number of bytes remaining
            // create and populate successive cacheNodes
        recsize /= 4; // number of dwords:
        while (recsize > 0)
          { pRes -= 4; // move to current dword
            stuff = *((Int32*)(pRes)); // [endian]
            cn = NewIdNode(refnum, stuff);
            cn->next = tnode->data;
            tnode->data = cn; // re-create cachenode chain
            recsize --;
          }; // We restore original node order!
        c_MemHandleUnlock(hmem);
       rec ++;
    };

  if (! c_DmCloseDatabase(storref))
     { return -5; // [ugh]
     };
  gl->ENABLED = 1; // (ensure) now can use!
  return 1; // success
}
```

4 Main functions

These are the actual caching functions. Also see section 1.5.

4.1 MAKECACHE

Given a particular table and relevant dependencies and a column, cache the desired rows, creating a p-table.

4.1.1 Subsidiary helper routines

First, create the given database, destroying old copy if present. Submit the actual table name, but prefix this with ”p-” and make the p-table. We then locate the database, returning the LOCAL ID (ugh) of the database, or zero on failure.

```
LocalID CacheMakeDatabase (Char * dname, Int16 dlen)
{
  LocalID lid;
  Char * p_name;

  if (! dname || dlen < 1)
     { return 0;
     };

  p_name = xNew(dlen+3);
  if (! p_name)
     { return 0;
     };
  xCopy(p_name+2, dname, dlen);
  xCopy (p_name, "p-", 2);
  * (p_name+dlen+2) = 0x0; // asciiz.

  if (c_DmCreateDatabase (p_name))
     { lid = c_DmFindDatabase (p_name); // slooow.
       Delete(p_name);
       return lid ; // success
     };

  // if failed, assume old database, delete and retry!
  lid = c_DmFindDatabase (p_name); // slooow.
  if (! c_DmDeleteDatabase(lid))
     { Delete(p_name);
       return 0; // blaagh.
     };
  if ( c_DmCreateDatabase (p_name))
     { lid = c_DmFindDatabase (p_name); // slooow.
```

```
       Delete(p_name);
       return lid;
     };
  Delete(p_name);
  return 0; // failed.
}
```

The following routine locates a source database, given the unadorned name (non-asciiz):

```
LocalID LocalFindDatabase (Char * dname, Int16 nlen )
{
  LocalID lid;
  Char * ascname;

  ascname = xNew(nlen+1);
  xCopy (ascname, dname, nlen);
  * (ascname+nlen) = 0x0;

  lid = DmFindDatabase(MAINCARD, ascname); // does database exist?
  Delete(ascname);

  return (lid);
}
```

Next, obtain the number of the target column, or return zero on failure. (The columns are in the same order in the header as in the data rows; we look for the column name **target** which has length **tlen** within the header which is contained in **sourcep**).

```
Int16 ReadInt16X (Char * mP)
{ return(   (*(mP+1) & 0xFF ) + ( (*(mP))<<8 )    );
}
```

NOTE: there was a problem in the header creation. Offsets of header 2 byte values seem to be being written to bad boundaries so we need to either fix this (later) or use a byte-sized read integer (use now).

There's also a problem with 32 bit reads from arbitrary fields. So the ugly solution (rewrite this) is:

```
Int32 ReadInt32X (UInt16 refnum, Char * myptr)
{
  Int32 i;
  i =   0xFF & ((Int32) *(myptr+3));
  i |= (0xFF & ((Int32) *(myptr+2)))<<8;
  i |= (0xFF & ((Int32) *(myptr+1)))<<16;
  i |= (0xFF & ((Int32) *(myptr+0)))<<24;
  return i;
}
```

(Noo. rather than Char * use UInt8 *. FIX ME).
Finally, obtain the number of the target column:

```
Int16 CacheTargetColumn(Char * sourcep, Int16 colCount,
                        Char * target, Int16 tlen)
{
  Int16 cc=0;
  Int16 targetColumn = 0;
  Int16 coloff;
  Int16 colname;
  Int16 collen;

  while (cc < colCount)
    { coloff = * ( (Int16 *) (sourcep+0x10 + 2*cc) );

      colname = ReadInt16X (sourcep+coloff); // REL offset
      collen =  (ReadInt16X (sourcep+coloff+2)); // length
      if (c_Same (sourcep+coloff+colname, collen, target, tlen) )
         {
           targetColumn = cc;
           cc = colCount; // force exit
         };
      cc ++;
    };
  return targetColumn;
}
```

4.1.2 Actual main routine

We test the main routine by pretending to submit a caching string in one of the following two formats. The first provides a particular identifier to 'slim down' a table:

```
CACHE(PROCESS.Person.1234)
```

We will ultimately allow submission of a set of values in braces, thus:

```
CACHE(PROCESS.Person.{1234,4567,4321})
```

. . . but at present only one value will be supplied. The second type of statement is similar, but provides a dependency:

```
CACHE(PROCESS:EPOCH.Process)
```

The table before the colon is the table depended *upon*, and the table after the colon is that (and the column) which depends on the preceding table. We might even remove the need for the 'stuff prior to the colon' but this provides a reasonable check.

The TableNode structure allows us to store primary keys identified in the first class of statement, but we should beware of having a vast list of such keys for two reasons, the first being speed, and the second more important one being memory constraints, although on modern PDAs local stack memory is less constrained than was the case with earlier versions of PalmOS! Clearly if we don't appropriately garbage-collect such nodes, we *will* run into problems.

The following (SUBCACHE) routine potentially accepts three strings, but occupancy of the pretable and person strings is (at least at present) mutually exclusive.

NOTE: We have a potential optimisation here. Terminal menus don't require cumbersome storage of hit nodes for later back references. If *clear*, the keeplink flag allows us to disable such storage.

```
Int16 SUBCACHE (UInt16 refnum,
   Char * pretable, Int16 prelen,
   Char * tbldotcolumn, Int16 tdclen,
   Char * person, Int16 perslen,
   Int16 keeplink )
{
  // 0. Initialise:
  TableNode * preNode = 0; // used in locating pretable
  TableNode * myNode = 0;
  Int16 ok=0;
  Int16 keep; // used to signal row needing storage.
  Int32 persn=0; // hmm

  LocalID srcdb; // ugh.
  DmOpenRef srcdata;
  LocalID destdb; // likewise.

  Int16 rowCount;
  Int16 rc;
  Char * sourcep;
  Int16 colCount;
  cacheNode * cnode;

  Char * colname;
  Int16 colnamelen;
  Int32 NEWROWS = 1; // 1 for the header!
  Int32 thiskey;
  Int16 mylen;
  Int16 err = 0; // error signal

  MemHandle sourcehand; //
  DmOpenRef destdata;
  MemHandle desthand;
  Char * destp;
  Int16 targetColumn;
  Int16 rowlen;
  Int16 coldatum;
  Int16 coldatlen;

//  Int16 DUDTEST=0;

  // ---------- end of header section ---------- //

   +OPTIONAL
     ConTx_c( refnum, "\nSUBCACHE: ", 11, 0);
     ConTx_c( refnum, tbldotcolumn, tdclen, 0);
  -OPTIONAL

  if (GetCaching(refnum)< 1) // if caching is disabled!
     { return -100;
     };
  if (   (! tbldotcolumn)
      || (tdclen < 3)
     )
     { return - 50; // nonsense.
     };
```

If there's a table depended on, then subsequent decisions will depend on values contained in the primary keys stored in the TableNode structure for that table. Otherwise, things are simpler.

```
if (   pretable
    && (prelen > 0)
   )
   { preNode = FindDatabaseByName (refnum, pretable, prelen);
     if (! preNode)
        { return -30; // fail
        };
   +OPTIONAL
     ConTx_c( refnum, "\nPre-node: ", 11, 0);
     ConTx_c( refnum, pretable, prelen, 0);
  -OPTIONAL
//  DUDTEST=1;
   } else // otherwise leave preNode zero.
   { // if NO pretable, must be 'person':
     if (! person)
```

```
         { return -31;
         };
      // read integer value for pers from person string:
      persn = c_Ascii2I32 (person, perslen);
      // value of -1 signals string error, and 0 is invalid.
      if (persn < 1)
         { return -32;
         };
    };
```

Isolate the current table name, and make a node. Before we make the node, create the actual database. We must also ensure that the source database exists, and open it!

```
  mylen = c_Advance ( tbldotcolumn, tdclen, '.'); // find TABLE.colname
  if (mylen < 2) //
     { return -2; // fail
     };
  colname = tbldotcolumn+mylen; // after the dot!
  colnamelen = tdclen-mylen;
  mylen --; // don't count the dot!
   +OPTIONAL
     ConTx_c( refnum, "\nCaching: ", 10, 0);
     ConTx_c( refnum, tbldotcolumn, mylen, 0);
  -OPTIONAL

  // locate the database:
  srcdb = LocalFindDatabase (tbldotcolumn, mylen); // slooow.
  if (! srcdb)
     { return -3;
     };
  // make p-database:
  destdb = CacheMakeDatabase(tbldotcolumn, mylen);
  if (! destdb)
     { return ok-20; // fail
     };
  // ... and the node: THE PROBLEM IS HERE!
  myNode = StoreDatabaseName (refnum, tbldotcolumn, mylen);
  if (! myNode)
     { return -4;
     };

//  if (DUDTEST)
//     { return -69;
//     };
```

Open the source database and obtain its characteristics, including the column details:

```
// open the database:
srcdata = c_DmOpenDatabase (srcdb, dmModeReadOnly);
if (! srcdata)
   {
     ObliterateTableNode (refnum, tbldotcolumn, mylen); // clumsy
     return -5;
   };
sourcehand = c_DmQueryRecord(srcdata, 0);  // read header
sourcep = (Char *) c_MemHandleLock(sourcehand);
rowCount = (Int16) (-(* ((Int32 *) (sourcep+8)))); // stored as -ve!
   // NB. ARM-incompatible (and all similar stmts).
// actual row count is stored at offset +8.
+OPTIONAL
 ConTx_c( refnum, " rows=", 6, 0);
 ConI_c (refnum, rowCount, 0);
-OPTIONAL
if (  (rowCount > MAXACTUALROWS)
    ||(rowCount < 0)
   )
   { ObliterateTableNode (refnum, tbldotcolumn, mylen); // clumsy
     return -9;
   };

   // Determine number of columns
colCount = * ( (Int16 *) (sourcep+0x0E) );
+OPTIONAL
 ConTx_c( refnum, ", cols=", 7, 0);
 ConI_c (refnum, colCount, 0);
-OPTIONAL

   // Find our target column
targetColumn = CacheTargetColumn(sourcep, colCount, colname, colnamelen);
if (! targetColumn)
   { // fail. Column not found.
     c_MemHandleUnlock(sourcehand);
     c_DmCloseDatabase(srcdata);
     ObliterateTableNode (refnum, tbldotcolumn, mylen); // clumsy
     return -6;
   };
 +OPTIONAL
   ConTx_c( refnum, " column: ", 9, 0);
   ConTx_c( refnum, colname, colnamelen, 0);
   ConTx_c( refnum, "[", 1, 0);
-OPTIONAL

   // Find length of header
rowlen = * ( (Int16 *) (sourcep+0x10 + 2*colCount) );
```

Now open the p-table (destination) database:

```
// Open the destination database p-PROCESS for writing
destdata = c_DmOpenDatabase (destdb, dmModeReadWrite);
if (! destdata)
   {
     c_MemHandleUnlock(sourcehand);
     c_DmCloseDatabase(srcdata);
     ObliterateTableNode (refnum, tbldotcolumn, mylen); // clumsy
     return -7;
   };
   // Write header record:
   ok = MakeFileRecord (refnum,
                        destdata, 0,
                        sourcep, rowlen);
   c_MemHandleUnlock(sourcehand); // done with this row.
   if (! ok) // fail
      { c_DmCloseDatabase (destdata);
        c_DmCloseDatabase(srcdata);
        ObliterateTableNode (refnum, tbldotcolumn, mylen); // clumsy
        return -8;
      };
```

Work through whole of source table, locating and copying relevant rows. Decision time. If **persn** (key value) is under 1, then use preNode to find relevant rows and store their primary keys in myNode->data. Otherwise use **persn** value to identify the keys.

```
rc = 1; // data row


// ------------------------------------------------
while ( rc < rowCount)
  {
    // Get current row
    sourcehand = c_DmQueryRecord(srcdata, rc);
    sourcep = (Char *) c_MemHandleLock(sourcehand);
    // Look up offset of column reference
    coldatum = * ( (Int16 *) (sourcep+0x10 + 2*targetColumn) );
    coldatlen= (* ( (Int16 *) (sourcep+0x12 + 2*targetColumn) )) - coldatum;
    ok = 1;
    keep = 0;
    if (persn > 0)
         {
           if ( c_Same( sourcep+coldatum, coldatlen, (Char*) (& persn), 4 ) )
              { keep = 1;
              };
         } else
         {
           thiskey = ReadInt32X (refnum, sourcep+coldatum);
```

```
          if ( FindIdNode(refnum, preNode->data, thiskey ) )
             { keep = 1;
             };
         };
      if (keep)
         { // if required row, then write to temporary database:
           +OPTIONAL
            ConTx_c( refnum, " ", 1, 0);
            ConI_c (refnum, rc, 0);
           -OPTIONAL
           rowlen = * ( (Int16 *) (sourcep+0x10 + colCount*2) );
           ok = MakeFileRecord (refnum,
                     destdata, (UInt16) NEWROWS,
                     sourcep, rowlen);
           NEWROWS ++; // only increment (e.g. 2) AFTER write!
           // 0 is header record, NEWROWS starts at 1.
           if (keeplink) // unless suppressed:
              { // retain process id in linked list:
                thiskey = * ( (Int32 *) (sourcep+8) );
                // is on 4 byte integral boundary, so ok.
                cnode = NewIdNode(refnum, thiskey); // new cache node
                cnode->next = myNode->data; //
                myNode->data = cnode;       // link into current TableNode
                +OPTIONAL
                 ConTx_c( refnum, "(", 1, 0);
                 ConI_c (refnum, thiskey, 0);
                 ConTx_c( refnum, ")", 1, 0);
                -OPTIONAL
              };
         };
      c_MemHandleUnlock(sourcehand); // ? check
      if (! ok) // fail
         {
           c_DmCloseDatabase (destdata);
           c_DmCloseDatabase(srcdata);
           ObliterateTableNode (refnum, tbldotcolumn, mylen); // clumsy
           return -40;
         };
      rc ++;
    }; // end of getting and writing relevant rows.
  // ------------------------------------------------

   // 3.5 Update header with NEWROWS value (-ve @ +8)
   NEWROWS = - NEWROWS;
   desthand = c_DmGetRecord(destdata, 0); // 1st rec, can write
   if (! desthand)
      { err = -20;
      } else
      { destp = (Char *) c_MemHandleLock(desthand); // lock
        c_DmWrite ( destp, 8, (Char *) (& NEWROWS), 4 );
        c_MemHandleUnlock(desthand);
        c_DmReleaseRecord (destdata, 0);
      };

   // 3.6 Close p-PROCESS.
   c_DmCloseDatabase(destdata);
   c_DmCloseDatabase(srcdata);
   if (err)
      { ObliterateTableNode (refnum, tbldotcolumn, mylen); // clumsy
        return err; // fail
      };
  +OPTIONAL
     ConTx_c( refnum, "]", 1, 0);
  -OPTIONAL

  return 1; // success.
}
```

4.1.3 The main MAKECACHE function

We submit a string in one of the formats previously referred to (Section 1.2). The table names *lack* a 'p-' prefix. The third parameter submitted to SUBCACHE was originally the id of a person (for the PERSON table) but despite now being more generic has retained the name! Note the potential for suppressing caching of keys by prefixing a table name with a tilde, something which disallows table names themselves from beginning with a tilde.

```
Int16 MAKECACHE (UInt16 refnum, Char * cachestring, Int16 cslen)
{
  Int16 ok;
  Char * pretable;
  Int16 prelen;
  Char * param2;
  Int16 p2len;
  Char * person;
  Int16 perslen;
  Int16 keeplink =1; // by default, keep.

  prelen = c_Advance(cachestring, cslen, ':');

  if (prelen < 2) // no pretable dependency:
      // format is thus "table.column.value":
     { prelen = c_Advance(cachestring, cslen, '.');
       param2 = cachestring+prelen;
       p2len = cslen - prelen;
       prelen = c_Advance(param2, p2len, '.');
       // go to after SECOND dot!
       if (prelen < 2)
          { return -3; // fail
          };
       person = param2 + prelen;
       perslen = p2len - prelen;
       param2 = cachestring; // !
       p2len = cslen - perslen -1;
         // -1 for the dot between initial stuff and 'person'.
       pretable = 0;
       prelen = 0;
     } else
       // format is TABLEa:TABLEb.column
       // variant is TABLEa:~TABLEb.column
     { pretable = cachestring;
       param2 = cachestring + prelen;
       p2len = cslen - prelen;
       if ((* param2) == '~')
          { keeplink = 0; //disable keep (terminal table)
            param2 ++;
            p2len --; // skip past the tilde
          };
       prelen --; // ignore the colon
       person = 0;
       perslen = 0;
     };

  ok = SUBCACHE (refnum, pretable, prelen,
                         param2, p2len,
                         person, perslen,
                         keeplink);
  if ( ok < 1 )
     {
       +OPTIONAL
        ConTx_c( refnum, "->", 2, 0);
        ConI_c (refnum, ok, 0);
       -OPTIONAL
       return -1;
     };
  return ok;
}
```

4.2 ISCACHING

Return the value in the ENABLED flag (0 or 1).

```
Int16 ISCACHING (UInt16 refnum)
{
  return GetCaching(refnum);
}
```

4.3 KILLCACHE

Delete all p-files. Return 1 on success, 0 on failure [???].

```
Int16 KILLCACHE (UInt16 refnum)
{
  return (KillAllTableNodes(refnum, 1));

  // hmm. later must be able to kill individual tables (but watch out for depender
}
```

4.3.1 Disable all caching

Returns 1 if on, 0 if off, -1 if error.

```
Int16 DisableAllCaching (UInt16 refnum)
{
  SysLibTblEntryPtr entryP = SysLibTblEntry (refnum);
  CacheLib_globals *gl = entryP->globalsP;
  if (!gl)
  {  return -1;
  };

  KILLCACHE(refnum); // nasty

  gl->ENABLED = 0;
  return 1;
}
```

4.4 DISABLECACHE

Turns off ability to cache! Simply wraps DisableAllCaching.

```
Int16 DISABLECACHE (UInt16 refnum)
{
  return DisableAllCaching(refnum);
}
```

4.5 CACHEFINDTABLE

Examine the name submitted. If it's one of the valid p-files and caching is enabled, then prepend 'p-' to the file name in the byref string provided. (There will be space). Otherwise do nothing. Return the *new length excluding the 0x0 AZCIIZ terminator* if changes were made, the old length (excluding ASCIIZ 0x0 terminator) if no changes.

At present we only prepend for the files EPOCH and PROCESS. Note that despite the length of the name being submitted, if the caller wishes for ASCIIZ, we supply a terminal 0x0 only if the name is altered, and do *not* write a terminal zero to the name if no 'p-' suffix is prepended.

```
Int16 CACHEFINDTABLE (UInt16 refnum, Char * tname, Int16 nlen)
{  // NOTE THAT nlen refers to the ACTUAL file name length, EXCLUDING an ASCIIZ te
   // IT IS ASSUMED that at least THREE characters exist above the top of the file

   // 0. Setup:
   Int16 test;
   Int16 i;

  +OPTIONAL
     ConTx_c( refnum, "\nFINDING:", 9, 0);
     ConTx_c( refnum, tname, nlen, 0);
  -OPTIONAL

// ==========================================================================

  // 1. Is caching on? If not, return the old length of the name.
  test = GetCaching(refnum);
  if (test != 1)
     { return nlen;
     };

  // 2. Is the file cached? If not, return with no changes:
  if (! FindDatabaseByName (refnum, tname, nlen))
     { return nlen;
     };

  // 3. prepend "p-" to the file name, shifting the name in the buffer.
  // It's important that the buffer is big enough. (See above).
  // We *also* ASCIIZ terminate the file, as a rather clumsy courtesy of
  //   debatable value.

  /// the following assumes that tname is in a writable buffer of correct size.
  i = nlen;
  while (i > 0)
    { i --;
```

```
      *(tname+i+2) = *(tname+i);
    };
  *(tname)='p';
  *(tname+1) = '-';     // prefix "p-"
  *(tname+nlen+2)= 0x0; // asciiz

  // 4. Return the new length of the name.
  +OPTIONAL
     ConTx_c( refnum, "\nFound: ", 8, 0);
     ConTx_c( refnum, tname, nlen+2, 0);
  -OPTIONAL

  return (nlen+2);

//  // ======================================================================

 return nlen;

}
```

4.6 CACHEINSERT

THE FOLLOWING HARD-CODING OF NAMES (EPOCH, PROCESS) *must* BE FIXED!!

Given a table name (**tname**) and its length, as well as a complete new line for that database (**dat**) and its length, insert the new line into the corresponding cache table. Return 0 if nothing done, 1 on success, and a negative number on serious failure.

The submitted table name is *NOT* the name of the cached file, so we still need to prepend the "p-" suffix!

We do *not* alter the cacheNode list attached to the TableNode for the table (which should ideally have been reclaimed before a CACHEINSERT is issued).

```
Int16 CACHEINSERT (UInt16 refnum, Char * tname, Int16 nlen,
                   Char * dat, Int16 datlen,
                   DmComparF * janet)
{

  // 0. setup:
  LocalID destdb;
  DmOpenRef destdata;
  Char * myname;
  UInt16 sortpos;
  Int32 NEWROWS;
  MemHandle desthand;
  Int16 ok;
  Char * destp;

  +OPTIONAL
     ConTx_c( refnum, "\nINSERTING:", 11, 0);
     ConTx_c( refnum, tname, nlen, 0);
  -OPTIONAL

// ===========================================================================

  // 0. is caching on? If not, exit, returning 0
  if (GetCaching(refnum) != 1)
     { return 0;
     };

  // 1. basic check:
  ok = 0;
  if (c_Same (tname, nlen, "PROCESS", 7))
     { ok = 1;
     } else
     { if (c_Same (tname, nlen, "EPOCH", 5))
          { ok = 1;
          };
     };
  if (! ok)
     { return 0;
     };

  // 2. modify name:
  myname = xNew(nlen+3);
  xCopy (myname+2, tname, nlen);
  xCopy (myname, "p-", 2);
  *(myname+nlen+2) = 0x0;

  // 3. Open the caching file
  destdb = c_DmFindDatabase (myname); // slooow.
  Delete(myname); // work is done.
  if (! destdb)
     { return -1; // not found.
     };
  destdata = c_DmOpenDatabase (destdb, dmModeReadWrite);
  if (! destdata)
     {
       return -2;
     };

  // 4. Our callback janet will use the correct datum at
  //    offset +8 in dat:
  sortpos = c_DmFindSortPosition (destdata, dat,
                                  0, janet, 0);

  // 5. insert the line.
  ok = MakeFileRecord ( refnum,
                        destdata, sortpos,
                        dat, datlen);
  // ??? should check on success

  // 6. update the record count in the header (record 0):
   desthand = c_DmGetRecord(destdata, 0); // 1st rec, can write
   destp = (Char *) c_MemHandleLock(desthand); // lock
   NEWROWS = * ( (Int32 *) (destp+8) ); // get -ve count
   NEWROWS --; // same as negate, increment, negate!
   c_DmWrite ( destp, 8, (Char *) (& NEWROWS), 4 );
   c_MemHandleUnlock(desthand);
   c_DmReleaseRecord (destdata, 0);

  // 7. close off:
  c_DmCloseDatabase(destdata);

// //============================================================================
 return 1; // success

}
```

It might later be feasible and wise to keep all caching files open within the library, to minimise PalmOs delays on find, open, close. (** NB **)

4.7 CACHEUPDATE

If a line has been altered in the source of a cached file, locate that line and make a similar update. Return 0 if not found, 1 if updated successfully, and -ve numbers with substantial errors. Confirms that we are writing to one of the two main files: p-EPOCH or p-PROCESS.

```
Int16 CACHEUPDATE (UInt16 refnum, Char * tname, Int16 nlen,
                   Char * dat, Int16 datlen,
                   DmComparF * janet)
{
  // 0. setup:
  LocalID destdb;
  DmOpenRef destdata;
  Char * myname;
  UInt16 sortpos;
  Int32 myKey;
  MemHandle desthand;
  Int16 ok=1;
  Char * destp;

  +OPTIONAL
     ConTx_c( refnum, "\nUPDATING:", 10, 0);
     ConTx_c( refnum, tname, nlen, 0);
  -OPTIONAL

// ===========================================================================
   // 0. If caching is off, exit returning 0.
  if (GetCaching(refnum) != 1)
     { return 0;
     };

  // 1. basic check:
  if (! FindDatabaseByName (refnum, tname, nlen))
     { return 0;
     };

  // 2. modify name:
  myname = xNew(nlen+3);
  xCopy (myname+2, tname, nlen);
  xCopy (myname, "p-", 2);
  *(myname+nlen+2) = 0x0;

  // 3. Open the caching file
  destdb = c_DmFindDatabase (myname); // slooow.
  Delete(myname); // work is done.
  if (! destdb)
     { return -1; // not found.
     };
  destdata = c_DmOpenDatabase (destdb, dmModeReadWrite);
  if (! destdata)
     {
       return -2;
     };

  // 4. Our callback janet will use the correct datum at
  //    offset +8 in dat:
  myKey = * ((Int32 *) (dat+8) ); // get value anyway!
  sortpos = c_DmFindSortPosition (destdata, dat,
                                0, janet, 0);

  // line should exist. If so, sortpos is pointing to NEXT line, thus:
  if ( sortpos < 1)
     {
       c_DmCloseDatabase(destdata);
       return -10;
     };
  sortpos --; // go to correct record.

  // 5. must still check that we have correct target:
   desthand = c_DmGetRecord(destdata, sortpos); //
   // might check for success?
   destp = (Char *) c_MemHandleLock(desthand); // lock
  if ( myKey != * ((Int32 *)(destp+8)) )
     {
       MemHandleUnlock(desthand);
       c_DmReleaseRecord (destdata, sortpos);
       c_DmCloseDatabase(destdata);
       return -11; // not found ??? aagh!
     };

  // 5. rewrite the line.
     c_MemHandleUnlock(desthand);
     desthand = c_DmResizeRecord(destdata, sortpos, datlen);
     if (! desthand)
        { c_DmCloseDatabase(destdata);
          return -12;
        };

     destp = (Char *)c_MemHandleLock(desthand); // resized
     ok = c_DmWrite( destp, 0, dat, datlen);

  // 7. close off:
  c_MemHandleUnlock(desthand);
  c_DmReleaseRecord (destdata, sortpos);
  c_DmCloseDatabase(destdata);

  // =========================================================================
  return 1; // success!
}
```

4.8 CACHEFLUSH

Given a particular table name, locate the cacheNode list for that table, and delete this linked list.

```
Int16 CACHEFLUSH (UInt16 refnum, Char * tname, Int16 nlen)
{
  TableNode * tn;
  Int16 hits;

  tn = FindDatabaseByName (refnum, tname, nlen);
  if (! tn)
     { return 0; // not found.
     };
  hits = KillAllIdNodes(refnum,tn->data);
  tn->data = 0;

  return hits;
}
```

4.9 UNCACHE

Remove a table (with all of its caching information) from the TableNode linked list.

```
Int16 UNCACHE (UInt16 refnum, Char * tname, Int16 nlen)
{
  return ObliterateTableNode (refnum, tname, nlen);
}
```

5 Header file: cache.h

We start with a few simple defines:

```
#ifndef CACHEEXTRA_H
#define CACHEEXTRA_H

#include <LibTraps.h>

#ifndef CACHE_TRAP
#define CACHE_TRAP(trapno)  SYS_TRAP(trapno)
#endif

// [fill in exports over here]
#define sysLibPassCacheBug       sysLibTrapCustom+0
#define sysLibTrapMAKECACHE      sysLibTrapCustom+1
#define sysLibTrapISCACHING      sysLibTrapCustom+2
#define sysLibTrapKILLCACHE      sysLibTrapCustom+3
#define sysLibTrapCACHEFINDTABLE sysLibTrapCustom+4
#define sysLibTrapCACHEINSERT    sysLibTrapCustom+5
#define sysLibTrapCACHEUPDATE    sysLibTrapCustom+6
#define sysLibTrapDISABLECACHE   sysLibTrapCustom+7
#define sysLibTrapCACHEFLUSH     sysLibTrapCustom+8
#define sysLibTrapUNCACHE        sysLibTrapCustom+9
#define sysLibTrapCACHESTORESTATE sysLibTrapCustom+10
#define sysLibTrapCACHESTATERESTORE sysLibTrapCustom+11
```

The actual routines, as seen externally.

```
Err CACHEOpen (UInt16 refnum)
    CACHE_TRAP(sysLibTrapOpen);

Err CACHEClose (UInt16 refnum, UInt16 *numappsP)
    CACHE_TRAP(sysLibTrapClose);

Int16 PassCacheBug (UInt16 refnum, UInt16 bugs, UInt16 errlib,
                    UInt16 console)
    CACHE_TRAP(sysLibPassCacheBug);

Int16 MAKECACHE (UInt16 refnum, Char * cachestring, Int16 cslen)
    CACHE_TRAP(sysLibTrapMAKECACHE);

Int16 ISCACHING (UInt16 refnum)
    CACHE_TRAP(sysLibTrapISCACHING);

Int16 KILLCACHE (UInt16 refnum)
    CACHE_TRAP(sysLibTrapKILLCACHE);
```

```
Int16 CACHEFINDTABLE (UInt16 refnum, Char * tname, Int16 nlen)
    CACHE_TRAP(sysLibTrapCACHEFINDTABLE);

Int16 CACHEINSERT (UInt16 refnum, Char * tname, Int16 nlen,
                   Char * dat, Int16 datlen, DmComparF * janet)
    CACHE_TRAP(sysLibTrapCACHEINSERT);

Int16 CACHEUPDATE (UInt16 refnum, Char * tname, Int16 nlen,
                   Char * dat, Int16 datlen, DmComparF * janet)
    CACHE_TRAP(sysLibTrapCACHEUPDATE);

Int16 DISABLECACHE (UInt16 refnum)
    CACHE_TRAP(sysLibTrapDISABLECACHE);

Int16 CACHEFLUSH (UInt16 refnum, Char * name, Int16 nlen)
    CACHE_TRAP(sysLibTrapCACHEFLUSH);

Int16 UNCACHE (UInt16 refnum, Char * name, Int16 nlen)
    CACHE_TRAP(sysLibTrapUNCACHE);

Int16 CACHESTORESTATE (UInt16 refnum)
    CACHE_TRAP(sysLibTrapCACHESTORESTATE);

Int16 CACHESTATERESTORE (UInt16 refnum)
    CACHE_TRAP(sysLibTrapCACHESTATERESTORE);

#endif
```

The end of the header file.

6 The Makefile

This is relatively straightforward, similar to other library makefiles such as that for the main scripting library.

```
LIBPATH = c:/palmdev/sdk-4
CREATOR = JxVS
LIBPATH = c:/palmdev/sdk-4
VERSION = 1

CC = m68k-palmos-gcc -Wall -g -O2 -mdebug-labels
AS = m68k-palmos-as

all: cache-syslib.prc

cache-syslib.prc: cache.def cache
build-prc -o $@ cache.def cache
ls -l *.prc

cache_objs = cache.o cache-dispatch.o

cache: $(cache_objs) Makefile
$(CC) -shared -nostartfiles -nostdlib -o $@ $(cache_objs) -lnfm -lgcc
m68k-palmos-objdump --section-headers cache

cache.o: cache.c cache.h

cache-dispatch.o: cache-dispatch.s

cache-dispatch.s: cache.def
m68k-palmos-stubgen cache.def

clean:
rm -f *.o *.prc *-dispatch.? cache
```

Appending the term ‘ -mdebug-labels’ to CC in the above makefile inserts names into the final code, permitting debugging and (notably) profiling.

7 The DEF file: cache.def

```
syslib { "Cache Library" CsQl }

export {
  CACHEOpen CACHEClose nothing nothing
  PassCacheBug
  MAKECACHE ISCACHING KILLCACHE CACHEFINDTABLE
  CACHEINSERT CACHEUPDATE DISABLECACHE
  CACHEFLUSH UNCACHE CACHESTORESTATE CACHESTATERESTORE
  }
```